

# An 8 Decimal Place PostScript Real Numeric Reporting Utility

**Don Lancaster**  
Synergetics, Box 809, Thatcher, AZ 85552  
copyright c2007 as **GuruGram #76**  
<http://www.tinaja.com>  
[don@tinaja.com](mailto:don@tinaja.com)  
(928) 428-4073

The **PostScript** language uses a 32-bit **IEEE floating point** precision which often approximates nearly 8 decimal places. But only **six** decimal places of precision are normally reported.

While adequate for the majority of PS users, certain **PostScript-as-Language** apps may demand (or at least welcome) more reportable precision.

The utility procs presented here provide reporting of **up to eight** decimal point precision. There typically will be a count or two of inaccuracy on the eighth decimal point. The reporting accuracy improvement will thus approach **100:1**. No internal mods to the PostScript interpreter are made. The reporting process is active only when specifically called.

Real numbers over a numeric range of **0.000000001** to **10,000,000** are handled. Larger numbers generate an error, while smaller ones truncate to zero.

The ready-to-use procs may be found [here](#).

## The Procs

We can start off with our usual **/mergestr** extraction from our **Gonzo Utilities...**

```
% /mergestr merges the two stack strings into one...  
/mergestr {2 copy length exch length add string dup dup  
4 3 roll 4 index length exch putinterval 3 1 roll exch 0 exch  
putinterval} def
```

A **/realto8dstring** is the high level code for the reporting conversion. It first determines the sign and then tests for values too large or too small. It then goes to **/processgoodreal** for actual report conversions. A final de-referencing is done to provide a unique output string...

```

/realto8dstring {dup 0 lt      % test and flag negatives
/isneg exch store
abs /val exch store         % save real as absolute value

val 100000000 ge {          % report and error trap values
(real is too big! )=      % that are too large to process
real_is_too_big! } if

val 0.00000001 le         % truncate smaller to zero
{{(0.0000000)}}
{processgoodreal} ifelse % process numbers versus zeros

isneg {(-)}{( )} ifelse   % create leading space or minus
exch mergestr            % add leading space or minus
20 string cvs           % dereference to avoid surprises
} store

```

Next, `/processgoodreal` continues the `realto8string` processing after numbers too large and too small have been dealt with. Subprocs are called for `tenmillions`, for `unitsormore`, and `fractions`. Note that `log floor cvi` tells you the `decade` size and position of any positive number...

```

/processgoodreal { val log % evaluate decimal location
floor cvi /posn exch store

posn 7 eq
{tenmillions} % treat ten millions special
{posn 0 ge
{unitsormore} % handle >1 as a class
{fractions} % handle fractions as a class
ifelse }
ifelse
} store

```

Support subproc `/tenmillions` handles ten millions as a special case needing no string reformatting...

```

/tenmillions { val round cvi % use ten millions val as is.
20 string cvs
} store

```

Support subproc **/unitsormore** handles units through millions...

```
/unitsormore {  
  /workstring val           % scale val to 10-99 megs  
  1 7 posn sub {10 mul}     % this may give more accuracy  
  repeat mul round  
  
  cvi 20 string cvs store   % and convert to string  
  
  workstring 0 posn 1 add    % stuff decimal point  
  getinterval (.) mergestr  
  
  workstring posn 1 add     % post remainder of string  
  workstring length 1 sub  
  posn sub getinterval  
  mergestr 20 string cvs   % dereference  
  } store
```

Support subproc **/fractions** handles fractional values...

```
/fractions {/workstring val % scale workstring  
  1 7 posn sub {10 mul}     % this may give more accuracy  
  repeat mul round  
  
  cvi 20 string cvs store   % and convert to string  
  
  (0.)posn neg 1 sub        % prepend leading zero and dp  
  {(0) mergestr} repeat    % add intermediate zeros  
  workstring mergestr     % postpend value  
  
  20 string cvs           % dereference  
  } store
```

## Some Examples

Here's a few examples to get you started. These exercise each and every possible decimal point position. The negative sign changes may be arbitrarily relocated...

```
2 sqrt 1000000 mul      realto8dstring ==  
2 sqrt 100000 mul neg  realto8dstring ==  
2 sqrt 10000 mul      realto8dstring ==  
2 sqrt 1000 mul neg   realto8dstring ==  
2 sqrt 100 mul       realto8dstring ==  
2 sqrt 10 mul neg    realto8dstring ==  
2 sqrt 10 mul       realto8dstring ==
```

```
2 sqrt 1      mul neg      realto8dstring ==
2 sqrt 0.1    mul          realto8dstring ==
2 sqrt 0.01   mul          realto8dstring ==
2 sqrt 0.001  mul          realto8dstring ==
2 sqrt 0.0001 mul          realto8dstring ==
2 sqrt 0.00001 mul         realto8dstring ==
2 sqrt 0.000001 mul        realto8dstring ==
2 sqrt 0.0000001 mul       realto8dstring ==
2 sqrt 0.00000001 mul      realto8dstring ==
2 sqrt 0.000000001 mul     realto8dstring ==
```

And you can uncomment this test for a "too large" error...

```
% 2 sqrt 100000000 mul      realto8dstring ==
```

### For Additional Assistance

Similar tutorials and additional support materials are found on our [PostScript](#), our [Math Stuff](#), and our [GurGram](#) library pages. As always, [Custom Consulting](#) is available on a cash and carry or contract basis. As are seminars.

For details, you can email [don@tinaja.com](mailto:don@tinaja.com). Or call (928) 428-4073.